Database Systems

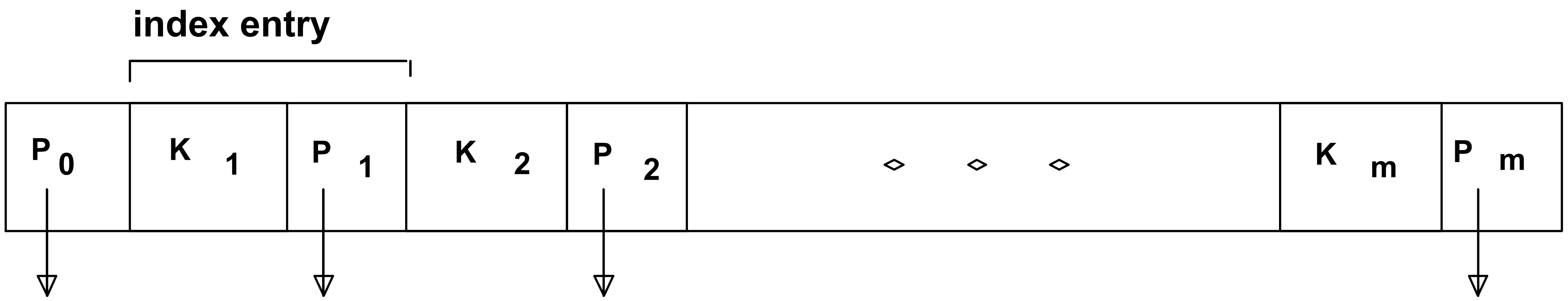
Index: B+ Tree

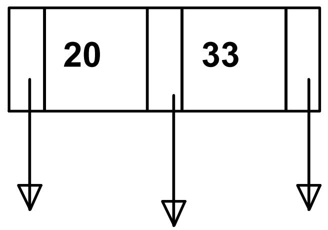
(the best data structure ever☺)

Based on slides by Feifei Li, University of Utah

Index Entries

An index entry has the following format: (search key value, page id). The following shows an index page with m index entries (pay attention to the special "left-most pointer")





Tree-based Indexes

- *Recall: 3 alternatives for data entries* k*:
  - Data record with key value **k**
  - <**k**, rid of data record with search key value **k**>
  - <**k**, list of rids of data records with search key **k**>
- Choice is orthogonal to the *indexing technique* used to locate data entries k*.
- Tree-structured indexing techniques support both *range searches* and *equality searches*.
- *ISAM*: static structure; *B+ tree*: dynamic, adjusts gracefully under inserts and deletes.
- ISAM = ???

Indexed Sequential Access Method

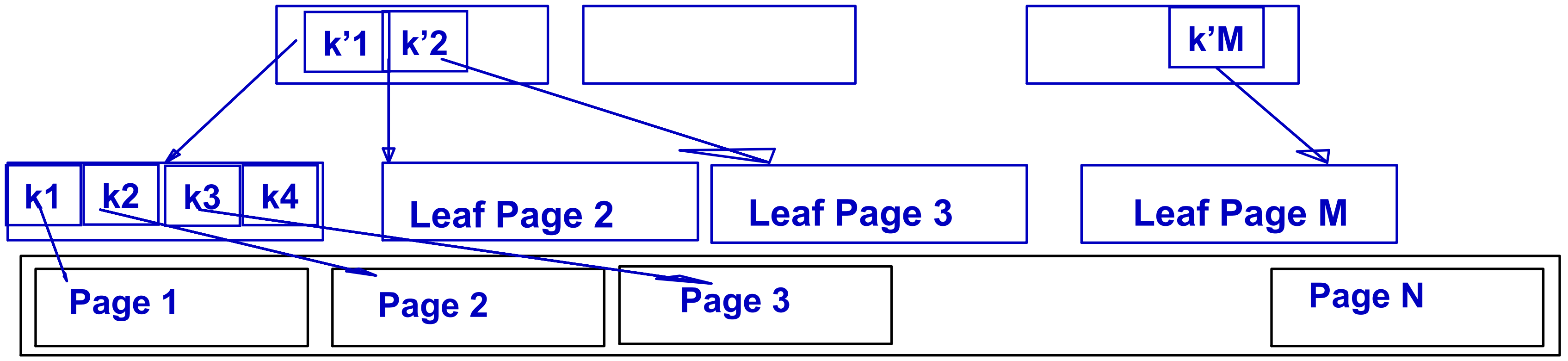
A Note of Caution

- ISAM is an old-fashioned idea
  - B+-trees are usually better, as we'll see
- But, it's a good place to start
  - Simpler than B+-tree, but many of the same ideas
- Upshot
  - **Don't** brag about being an ISAM expert on your resume
  - **Do** understand how they work, and tradeoffs with B+-trees

Range Searches

- ``*Find all students with gpa > 3.0*''
  - If data is in sorted file, do binary search to find first such student, then scan to find others.
  - Cost of binary search can be quite high.
- Simple idea: Create an `index' file.
  - Level of indirection again!

Index File:
Take the smallest search key value from each leaf page to build the index entries!

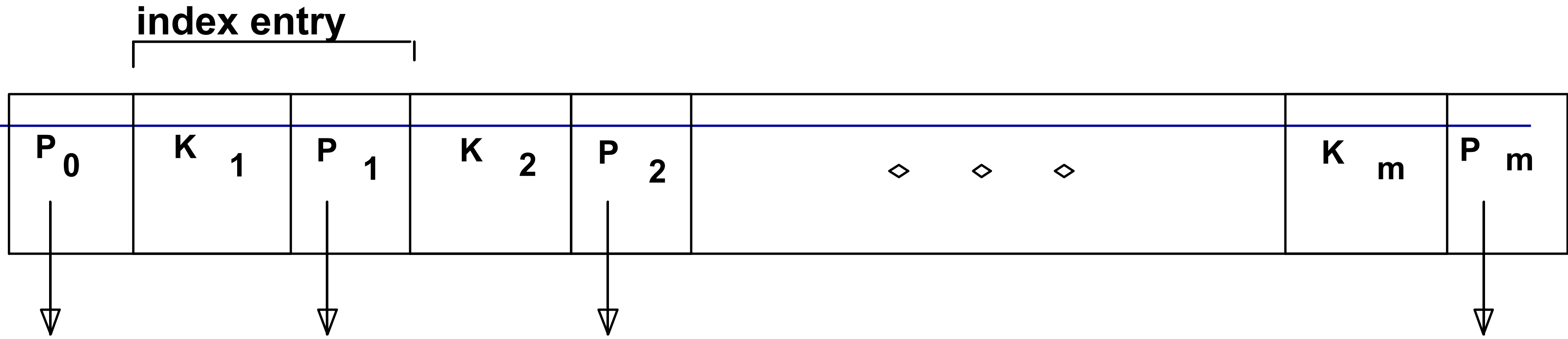


Leaf Pages with Data Entries:
1) One data entry per record!
2) Sort data entries

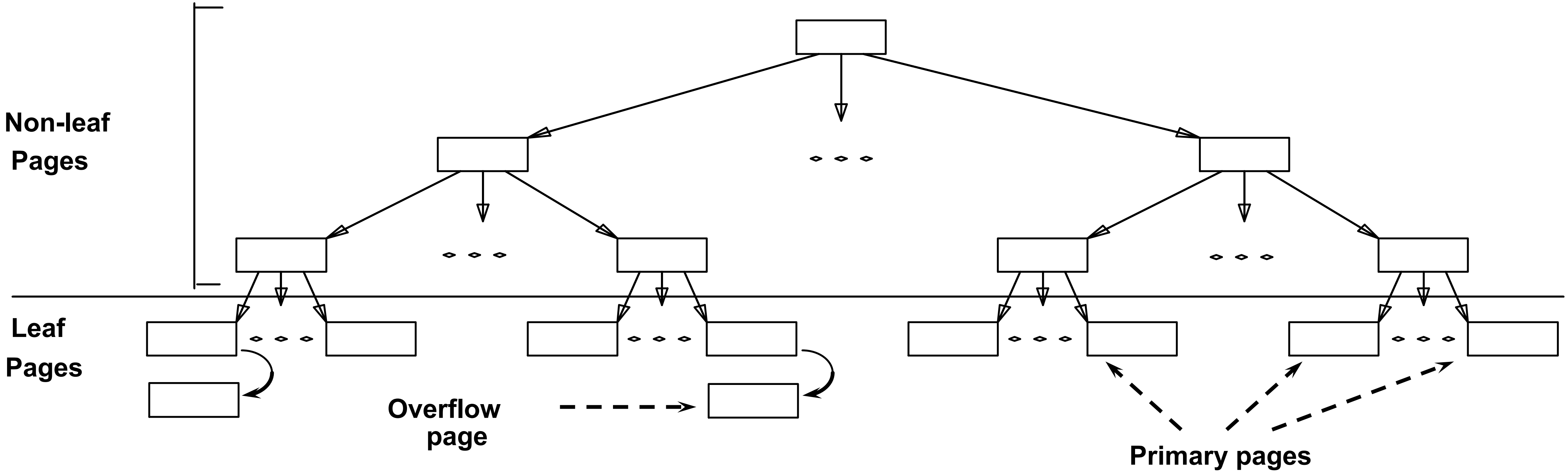
Data File
With Data Pages

☛ ***Can do binary search on (smaller) index file!***

ISAM



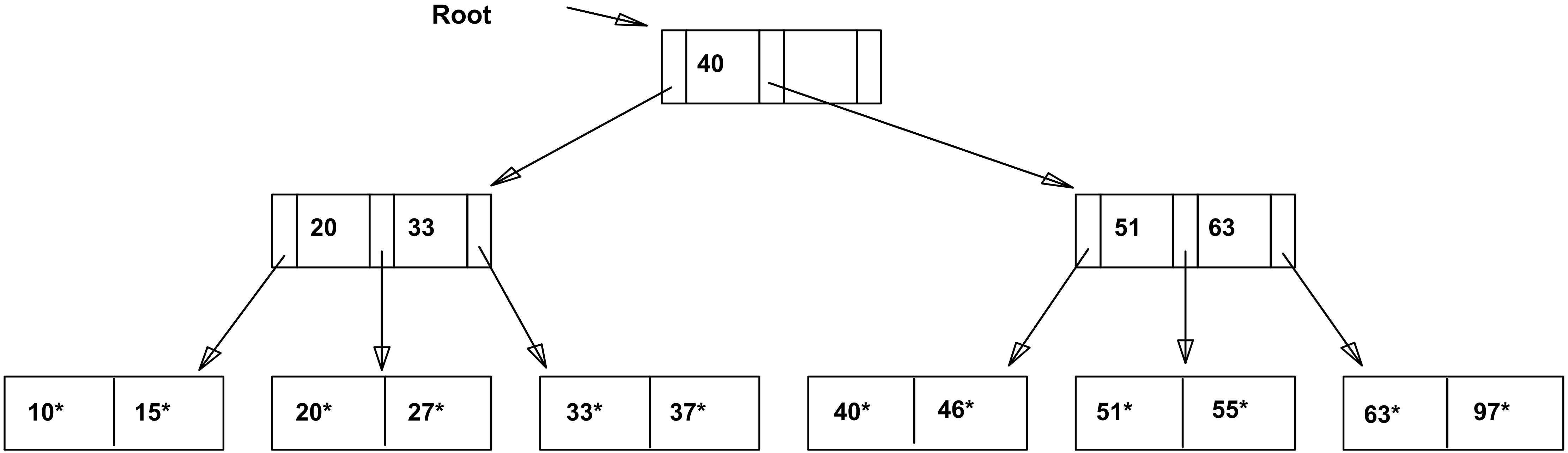
- Index file may still be quite large. But we can apply the idea repeatedly!



☛ ***Leaf pages contain data entries.***

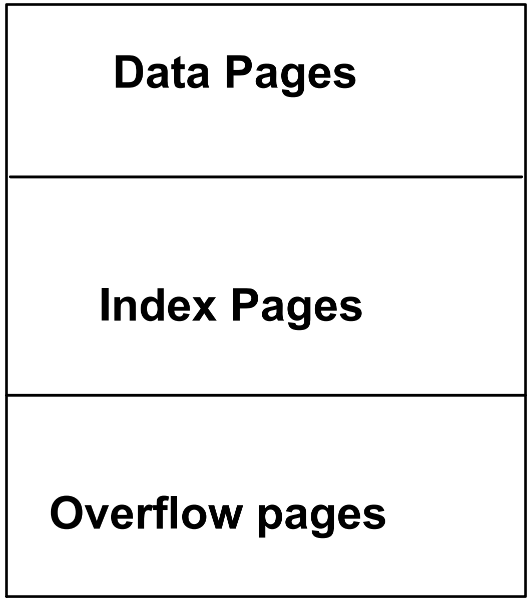
Example ISAM Tree

- Each node can hold 2 entries; no need for `next-leaf-page’ pointers. (Why?)



Comments on ISAM

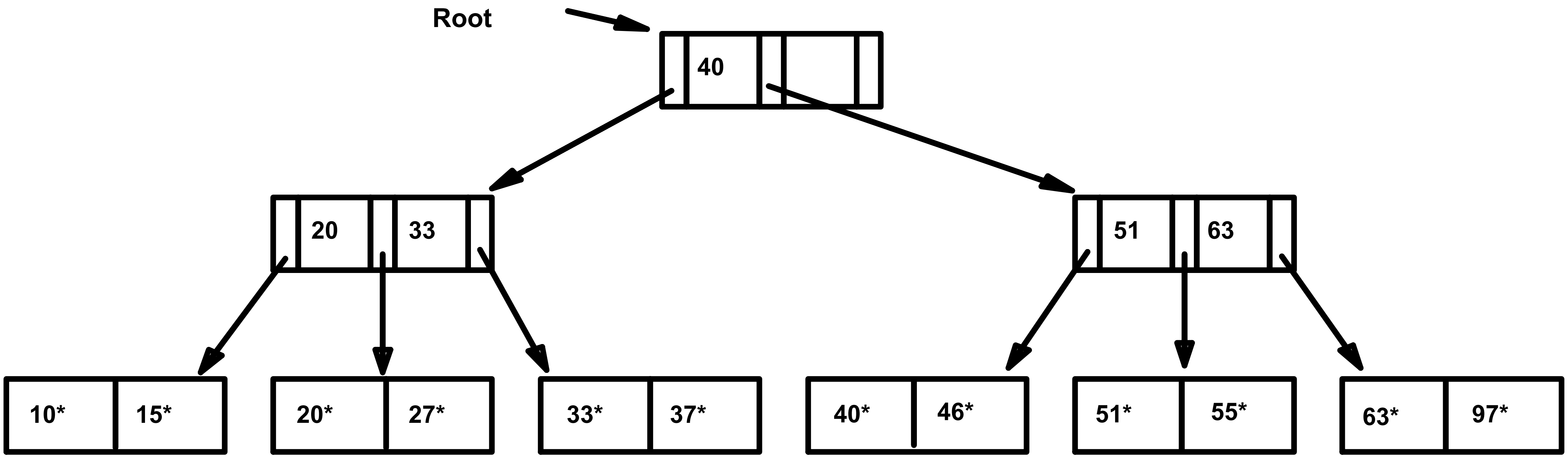
- *File creation*: Data pages first. Leaf (data) pages allocated sequentially, sorted by search key.
  Then index pages allocated.
  Then space for overflow pages.
- *Index entries*: <search key value, page id>; they `direct' search for *data entries*, which are in leaf pages.
- <u>*Search*</u>: Start at root; use key comparisons to go to leaf. Cost $\propto \log_F N$; F = # entries per index page, N = # leaf pages
- <u>*Insert*</u>: Find leaf where data entry belongs, put it there. (Could be on an overflow page).
- <u>*Delete*</u>: Find and remove from leaf; if empty overflow page, de-allocate.



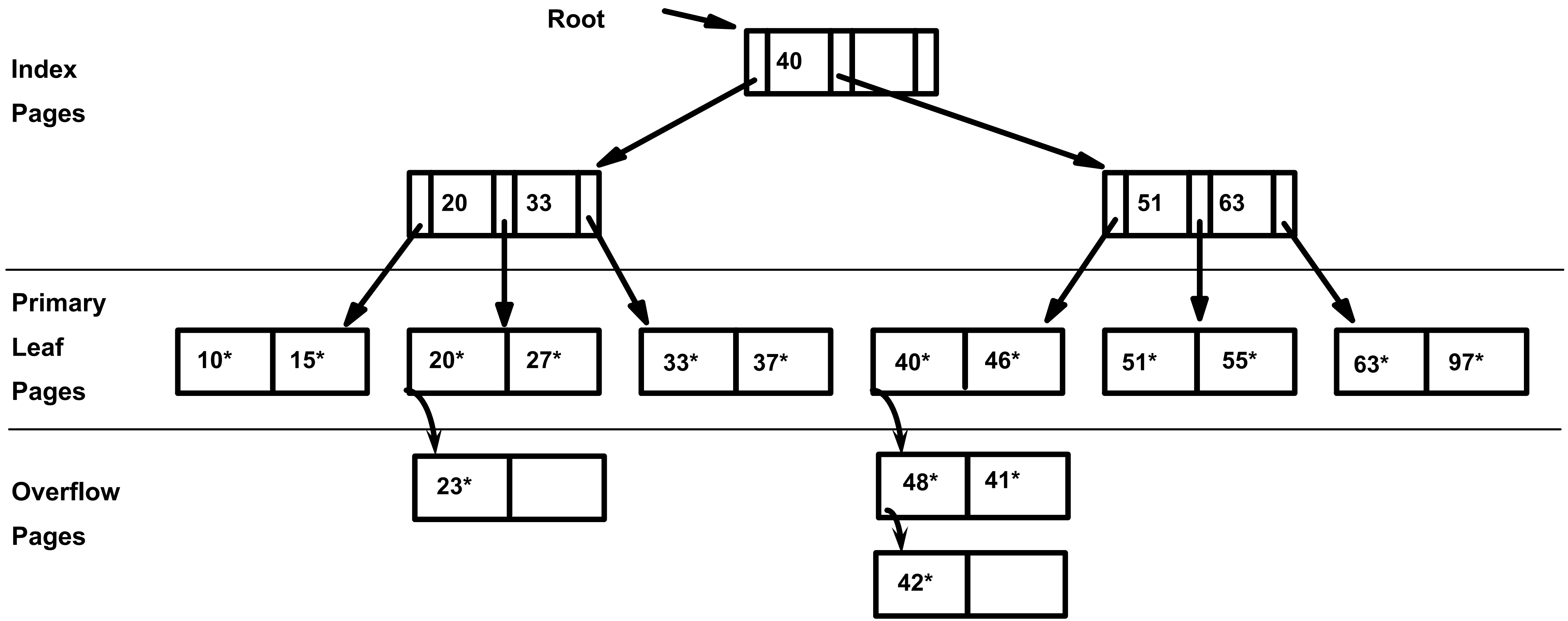
☛ **Static tree structure: *inserts/deletes affect only leaf pages*.**

Example ISAM Tree

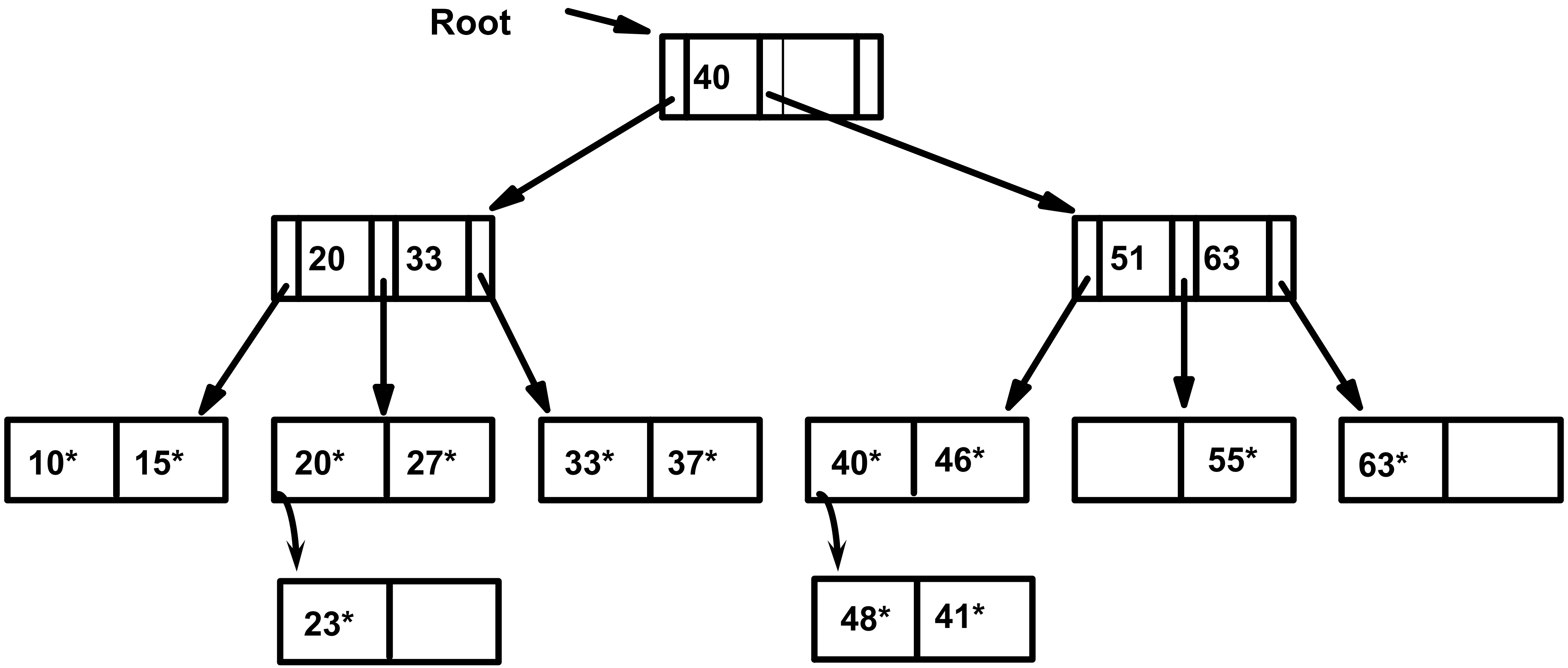
- Each node can hold 2 entries; no need for `next-leaf-page' pointers. (Why?)



After Inserting 23*, 48*, 41*, 42* ...



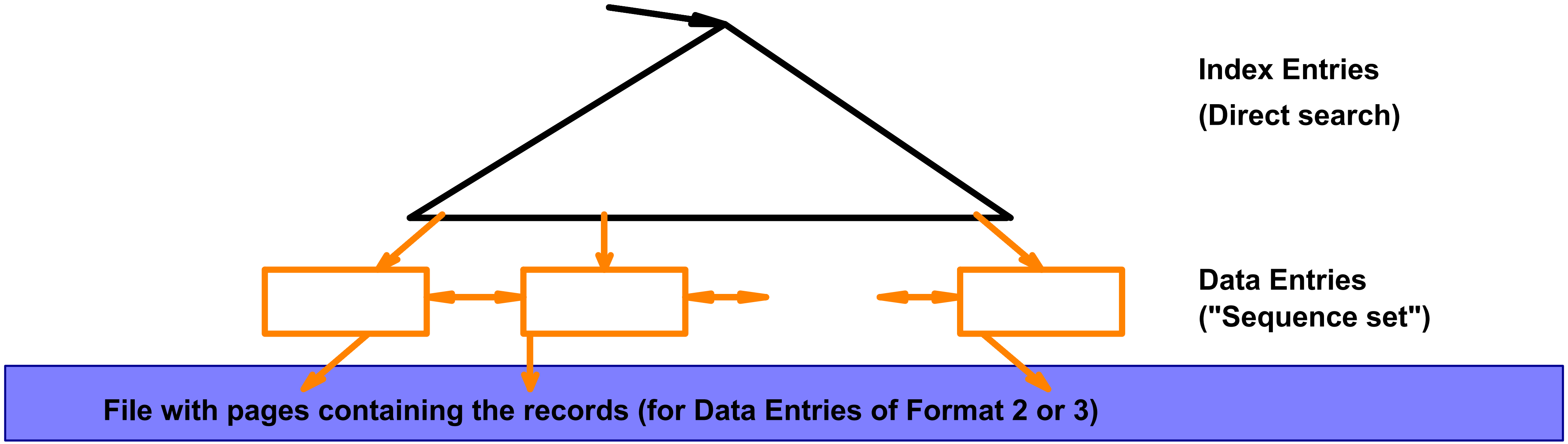
... then Deleting 42*, 51*, 97*



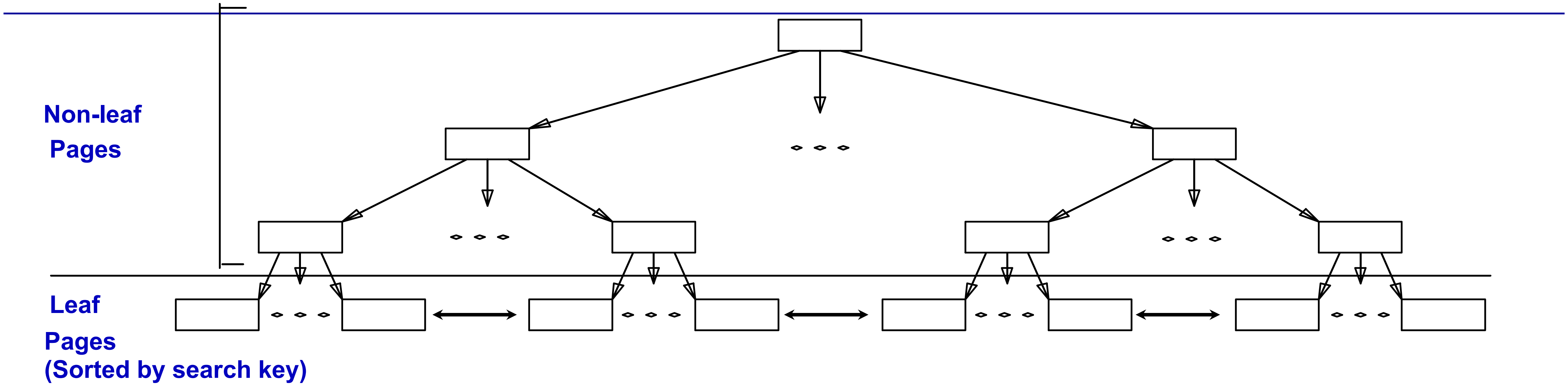
☛ ***Note that 51 appears in index levels, but 51* not in leaf!***

B+ Tree: The Most Widely Used Index

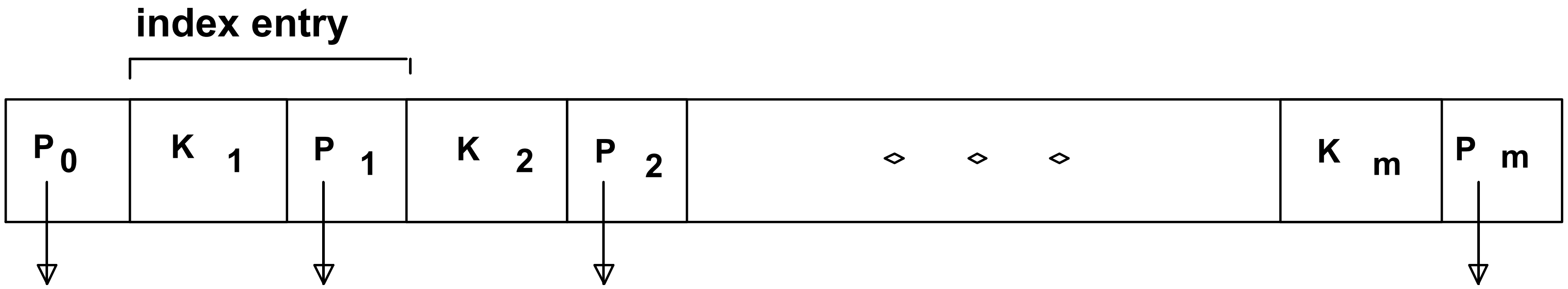
- Insert/delete at $\log_F N$ cost; keep tree *height-balanced*.
  (F = fanout, N = # leaf pages)
- Minimum 50% occupancy (except for root). Each node contains d <= *m* <= 2d entries. The parameter d is called the *order* of the tree.
- Supports equality and range-searches efficiently.



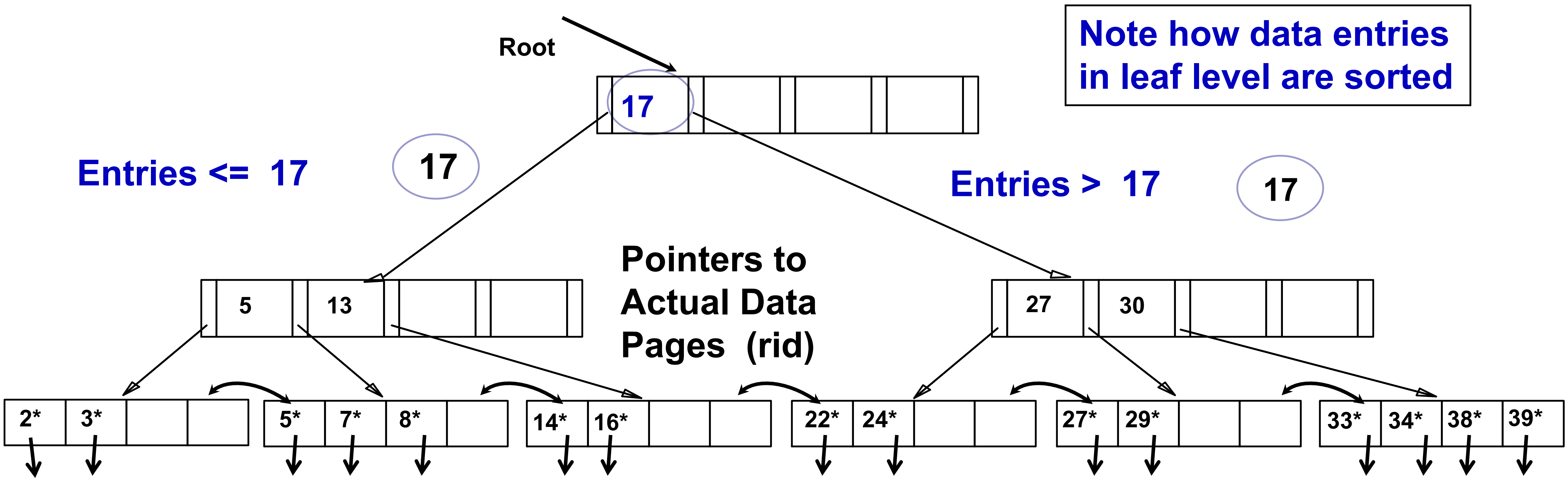
B+ Tree Indexes



- Leaf pages contain *data entries*, and are chained (prev & next)
- Non-leaf pages have *index entries;* only used to direct searches:



Example B+ Tree



- Find 28*? 29*? All > 15* and < 30*
- Insert/delete: Find data entry in leaf, then change it. Need to adjust parent sometimes.
  - And change sometimes bubbles up the tree

Analysis of B+ Tree

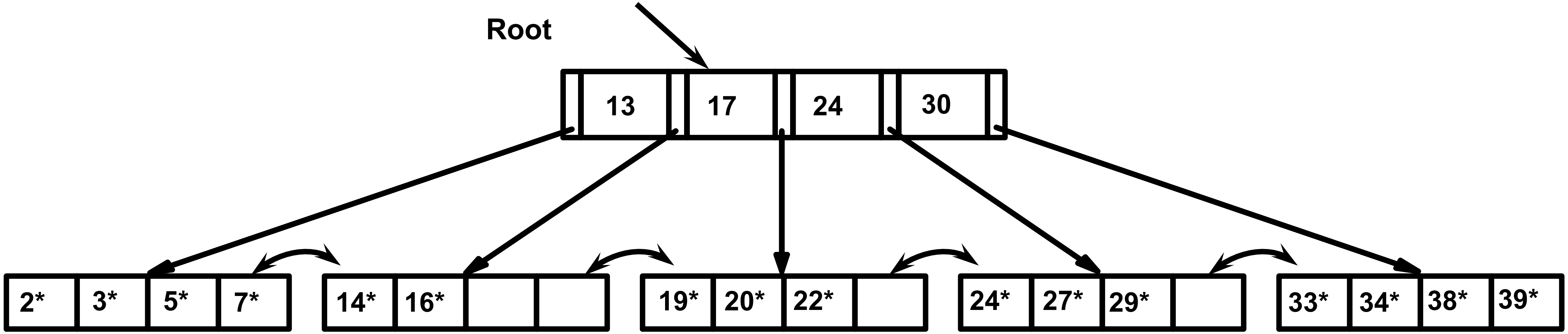
- Suppose Page size is P (bytes), each record is r (bytes), search key is 4 bytes, each pointer/record id/page id is 4 bytes, and N records in total, **alt 2** is used for a data entry.
- Bottom-up analysis:
  - Number of pages in the **data** file: $M = N / \lfloor P/r \rfloor$
    - Example: N=1M, P=4kbytes, r=100 bytes => P/r =40, M= 1M/40 = 25000
  - Number of data entries: N (one per record)
  - Size of a data entry: 8 bytes
  - Number of pages in leaf level:
    - $N' = N / \lfloor P/8 \rfloor$

Analysis of B+ Tree (contd)

- Index Level:
  - Number of index entries per page: f=((P-4)/8)*u (u is the average utilization ratio: [0.5, 1])
  - Number of entries in the index level right above the leaf level: N' (one entry per leaf-level page)
  - Number of pages required in this level: N'/f
  - Number of entries in the level above: N'/f
  - Number of pages in the level above: N'/f^2
  - Recursively pages in each level:
    - N', N'/f, N'/f^2 , N'/f^3 …. $1=N'/f^h$
    - So $h=\log_f N'$ (the height of the tree will be h or h+1 depending if you count the root level or not), total number of pages N'+N'/f+…+1=O(N')

Example B+ Tree

- Search begins at root, and key comparisons direct it to a leaf (as in ISAM).
- Search for 5*, 15*, all data entries >= 24* ...



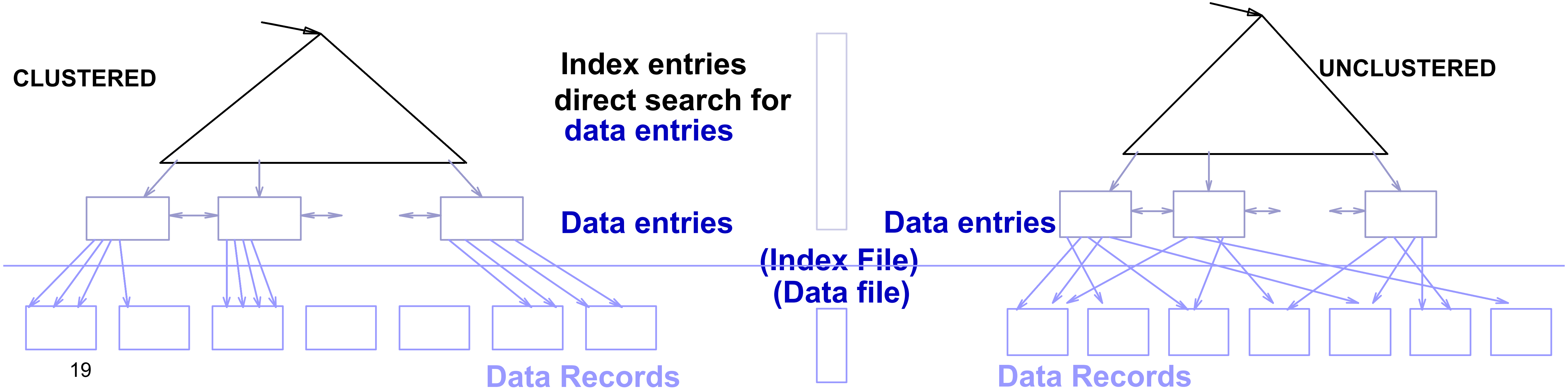
☛ ***Based on the search for 15*, we <u>know</u> it is not in the tree!***

Index Classification

- *Clustered* vs. *unclustered*: If order of data records is the same as, or `close to', order of index data entries, then called *clustered index*.
  - A file can be clustered on at most one search key.
  - Cost of retrieving data records through index varies *greatly* based on whether index is clustered or not!
  - Alternative 1 implies clustered, *but not vice-versa*.

Clustered vs. Unclustered Index

- Suppose that Alternative (2) is used for data entries, and that the data records are stored in a Heap file.
  - To build clustered index, first sort the Heap file (with some free space on each block for future inserts).
  - Overflow blocks may be needed for inserts. (Thus, order of data recs is `close to', but not identical to, the sort order.)



Unclustered vs. Clustered Indexes

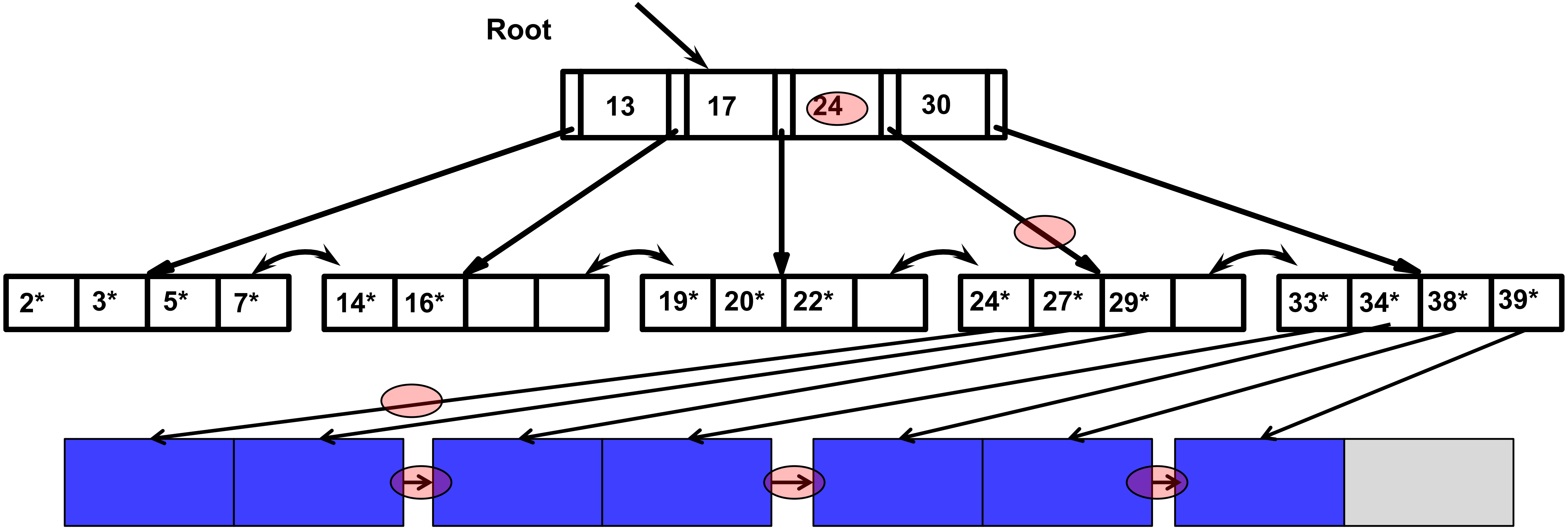
- What are the tradeoffs????
- Clustered Pros
  - Efficient for range searches
  - May be able to do some types of compression
  - Possible locality benefits (related data?)
- Clustered Cons
  - Expensive to maintain (on the fly or sloppy with reorganization)

Clustered Files

- We usually refer a clustered Index using Alternative 1 as clustered files, i.e., data entries in the leaf-level are records themselves! Data File itself becomes your level pages.
- Pages are usually about 67 percent occupancy
  - No. of physical data pages is about 1.5N/B (if N/B pages is required for storing all the data when each page is fully utilized)

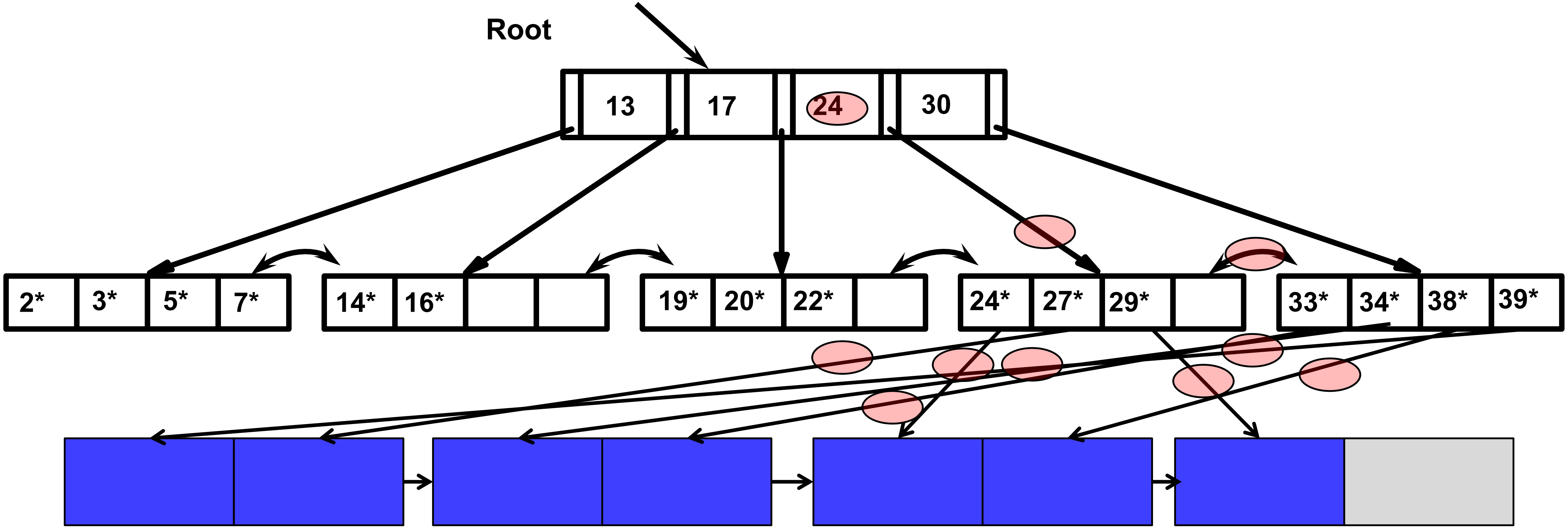
Example of B+ Tree (contd)

- All records >= 24. Clustered Index. 6 IOs



Example of B+ Tree (contd)

- All records >= 24. Unclustered Index: 10 IOs



B+ Tree in MySQL Continued.

- Now try the same queries with a tree-index built.
  - CREATE [UNIQUE|FULLTEXT|SPATIAL] INDEX ***index_name*** [***index_type***]

    ON ***tbl_name*** (***index_col_name***,...)

    [***index_type***]

index_col_name: ***col_name*** [(***length***)] [ASC | DESC]

index_type: USING {BTREE | HASH}

- Many engines create a clustered index on your primary key automatically.

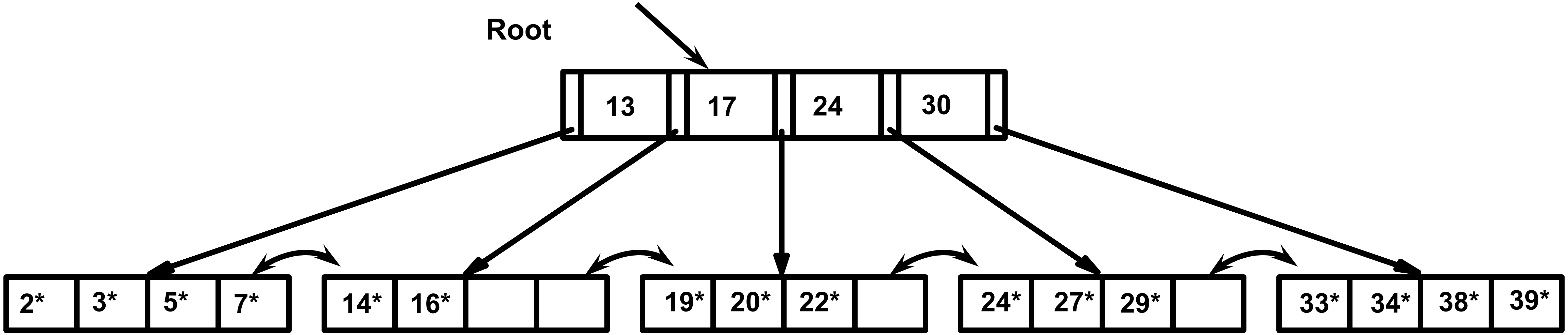
B+ Trees in Practice

- Typical order: 100 (B = 200). Typical fill-factor: 67%.
  - average fanout = 133
- Typical capacities:
  - Height 4: 133^4 = 312,900,700 records
  - Height 3: 133^3 = 2,352,637 records
- Can often hold top levels in buffer pool (in almost all systems, root level will always be buffered):
  - Level 1 = 1 page = 8 Kbytes
  - Level 2 = 133 pages = 1 Mbyte
  - Level 3 = 17,689 pages = 133 MBytes

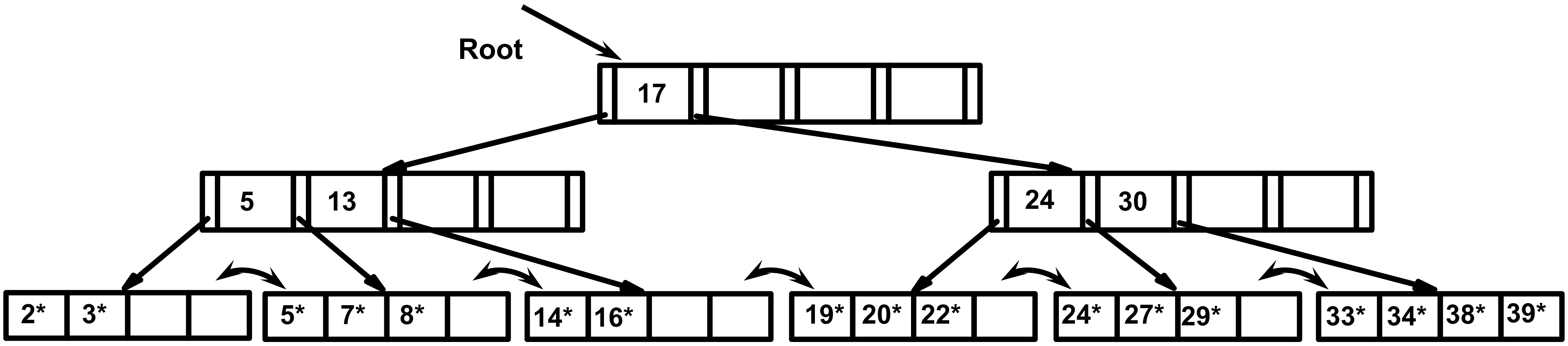
Inserting a Data Entry into a B+ Tree

- Find correct leaf *L.*
- Put data entry onto *L*.
  - If *L* has enough space, *done*!
  - Else, must *split* *L (into L and a new node L2)*
    - Redistribute entries evenly, **copy up** middle key.
    - Insert index entry pointing to *L2* into parent of *L*.
- This can happen recursively
  - To split index node, redistribute entries evenly, but **push up** middle key. (Contrast with leaf splits.)
- Splits "grow" tree; root split increases height.
  - Tree growth: gets *wider* or *one level taller at top.*

Example B+ Tree - Inserting 8*



Example B+ Tree - Inserting 8*

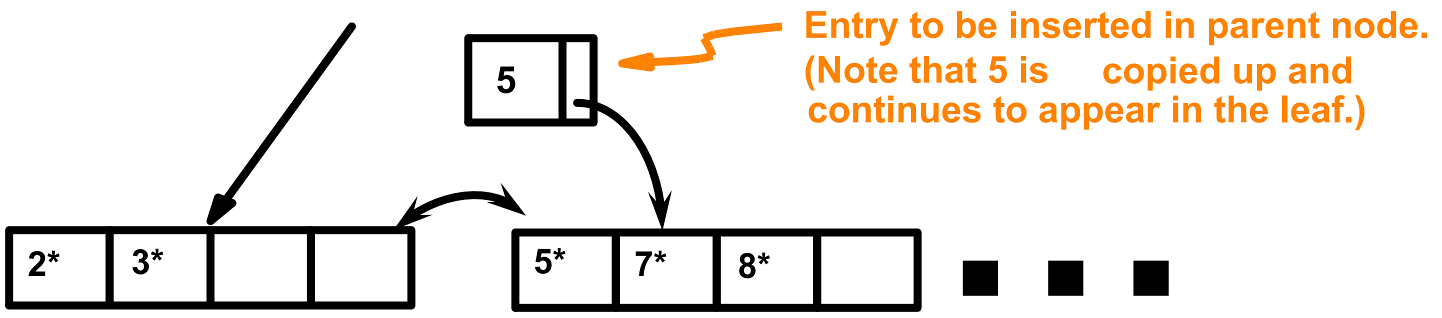


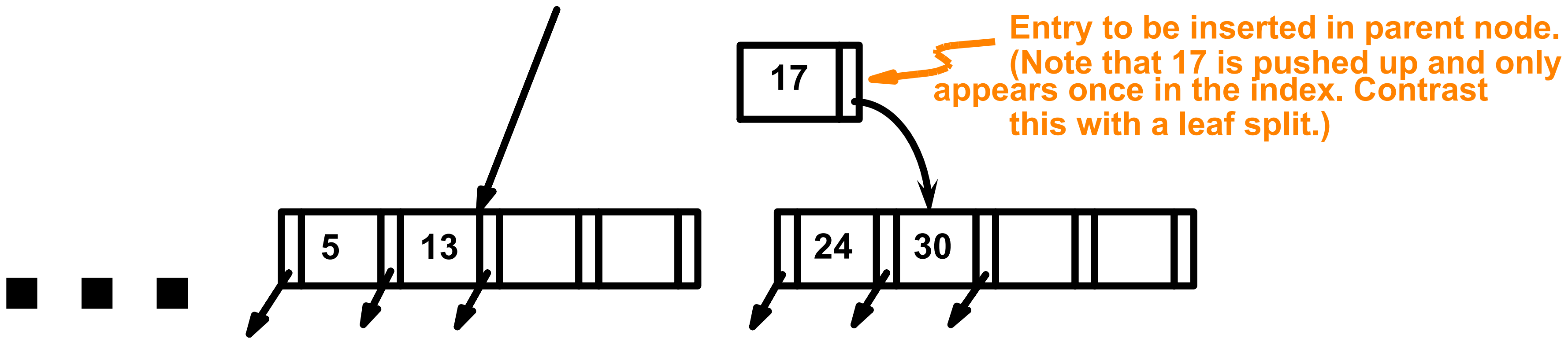
❖ **Notice that root was split, leading to increase in height.**

❖ **In this example, we can avoid split by re-distributing entries; however, this is usually not done in practice.**

Inserting 8* into Example B+ Tree

- Observe how minimum occupancy is guaranteed in both leaf and index pg splits.
- Note difference between *copy-up* and *push-up*; be sure you understand the reasons for this.

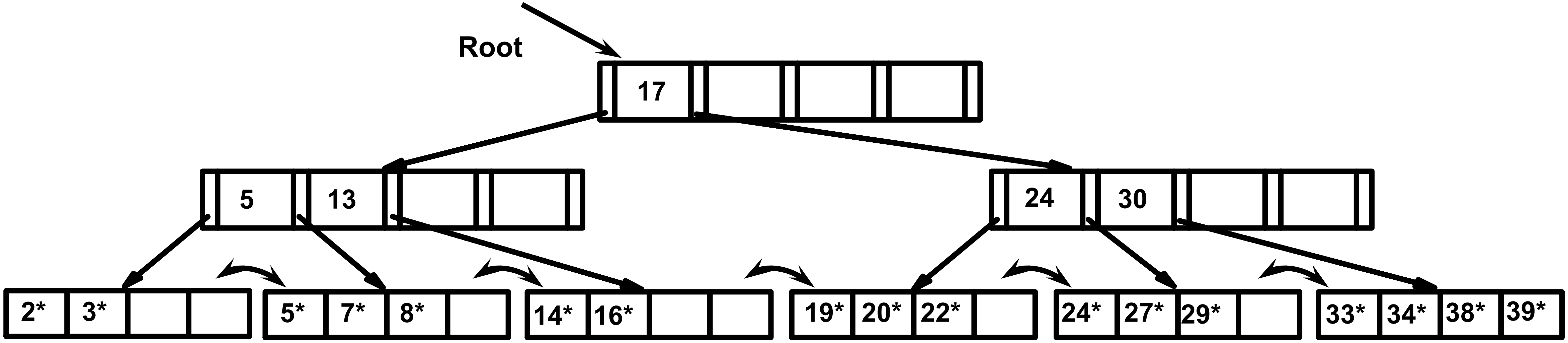




Deleting a Data Entry from a B+ Tree

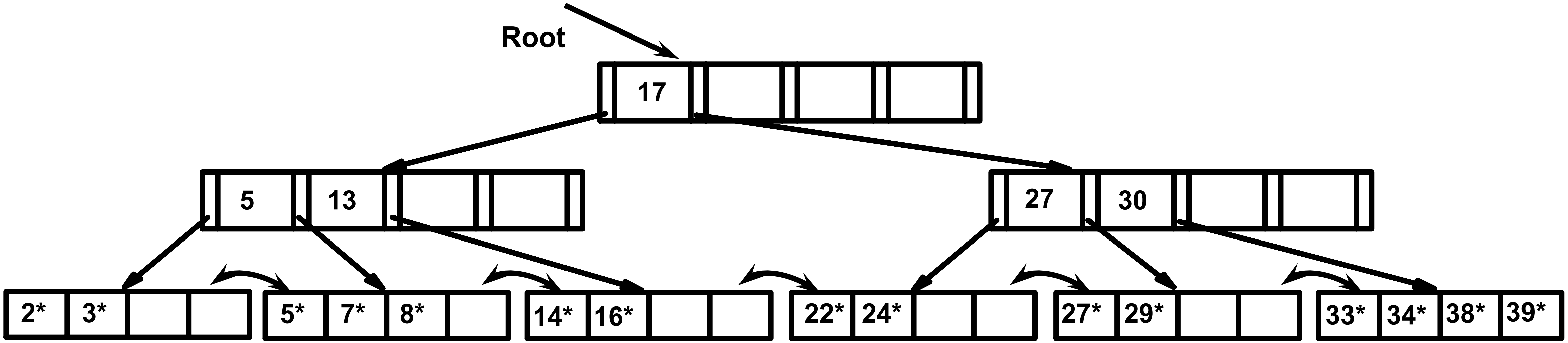
- Start at root, find leaf *L* where entry belongs.
- Remove the entry.
  - If L is at least half-full, *done!*
  - If L has only **d-1** entries,
    - Try to re-distribute, borrowing from *sibling* *(adjacent node with same parent as L)*.
    - If re-distribution fails, *merge* *L* and sibling.
- If merge occurred, must delete entry (pointing to *L* or sibling) from parent of *L*.
- Merge could propagate to root, decreasing height.

Example Tree (including 8*) Delete 19* and 20* ...



- Deleting 19* is easy.

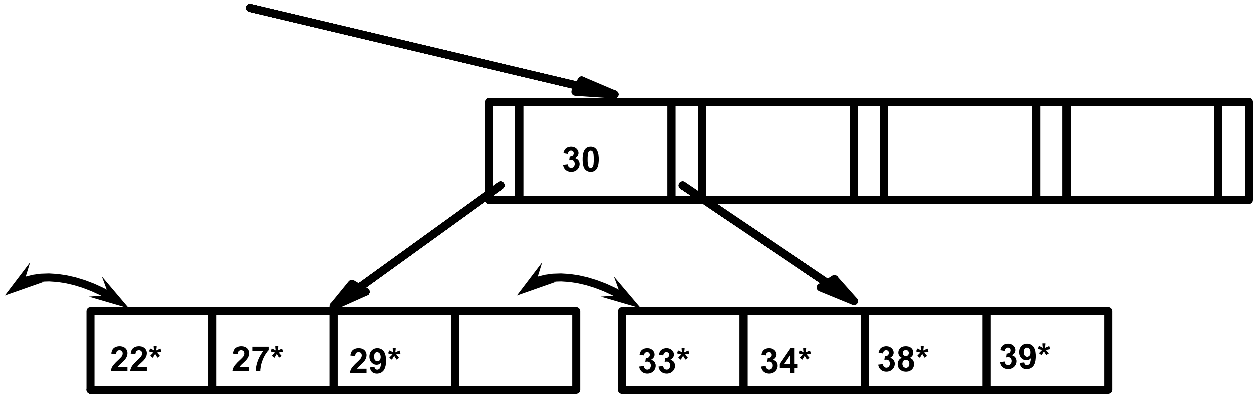
Example Tree (including 8*) Delete 19* and 20* ...

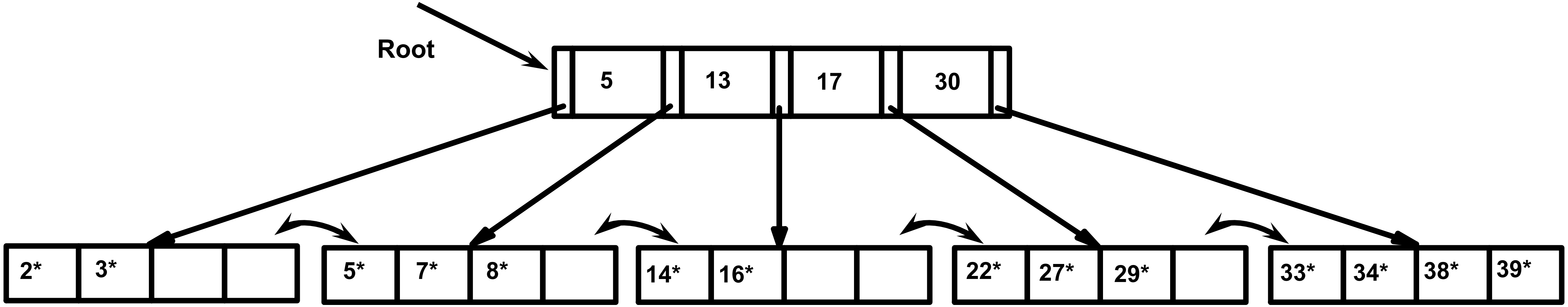


- Deleting 19* is easy.
- Deleting 20* is done with re-distribution. Notice how middle key is *copied up*.

... And Then Deleting 24*

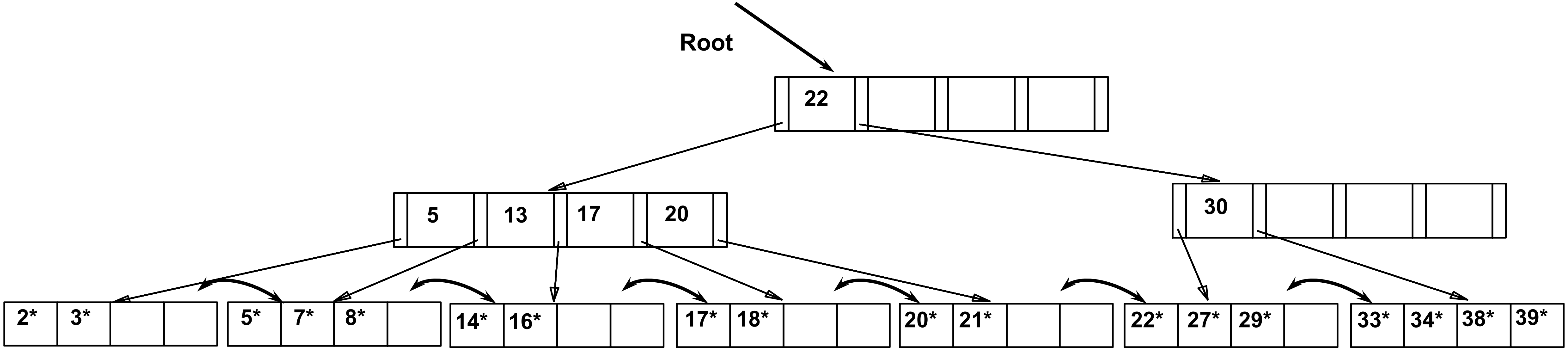
- Must merge.
- Observe *`toss'* of index entry (on right), and *`pull down'* of index entry (below).





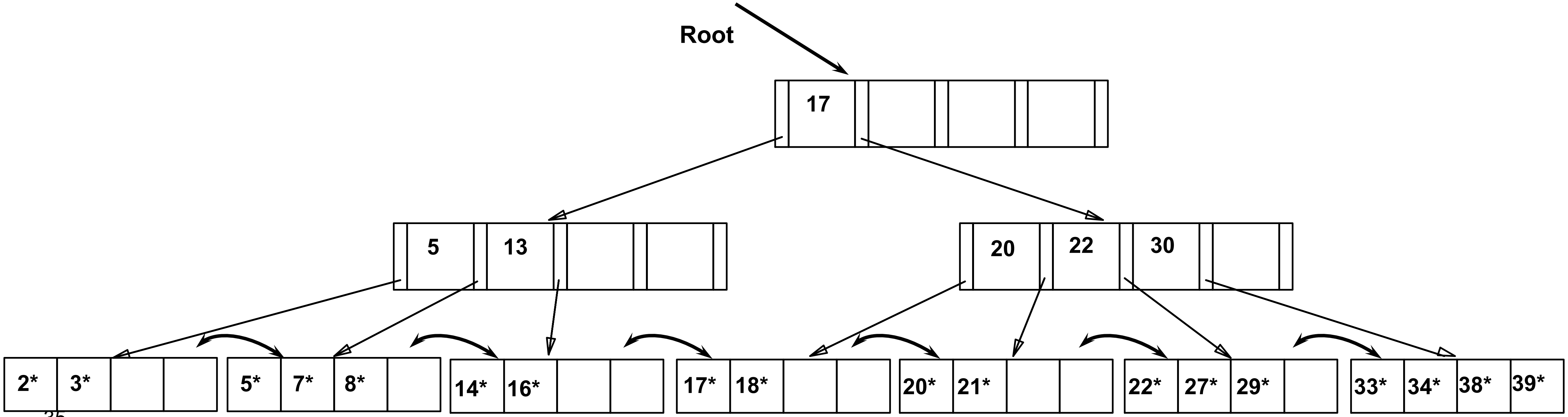
Example of Non-leaf Re-distribution

- Tree is shown below *during deletion* of 24*. (What could be a possible initial tree?)
- In contrast to previous example, can re-distribute entry from left child of root to right child.

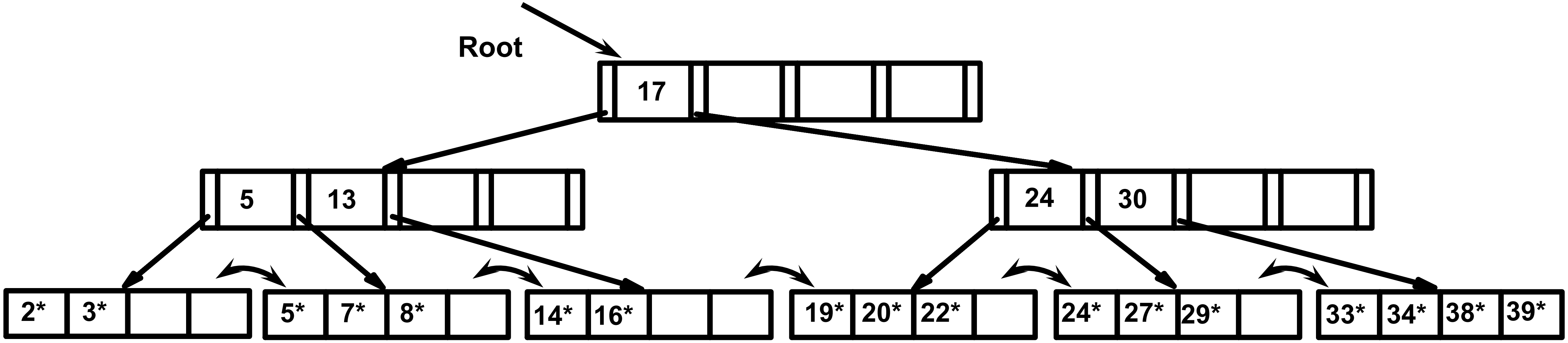


After Re-distribution

- Intuitively, entries are re-distributed by `*pushing through*' the splitting entry in the parent node.
- It suffices to re-distribute index entry with key 20; we've re-distributed 17 as well for illustration.

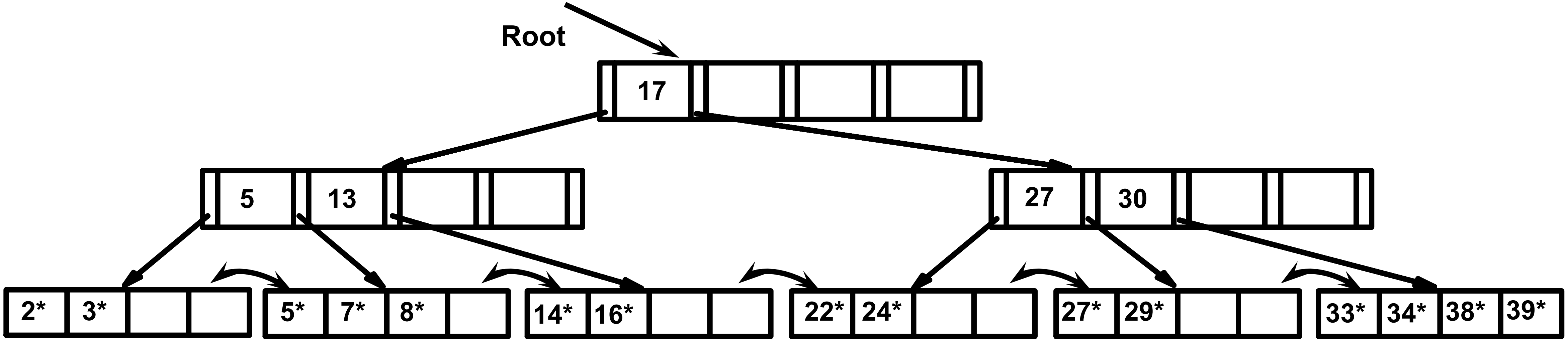


Example Tree (including 8*) Delete 19* and 20* ...



- Deleting 19* is easy.

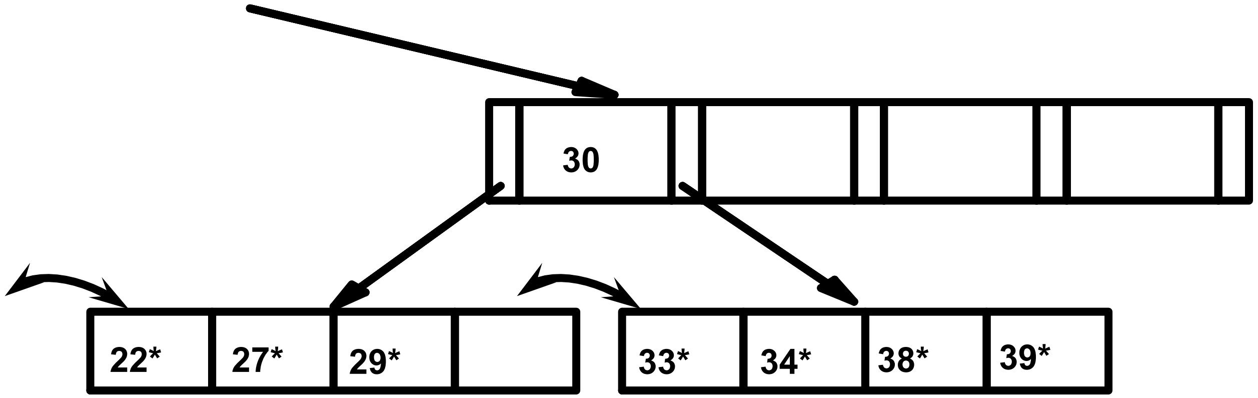
Example Tree (including 8*) Delete 19* and 20* ...

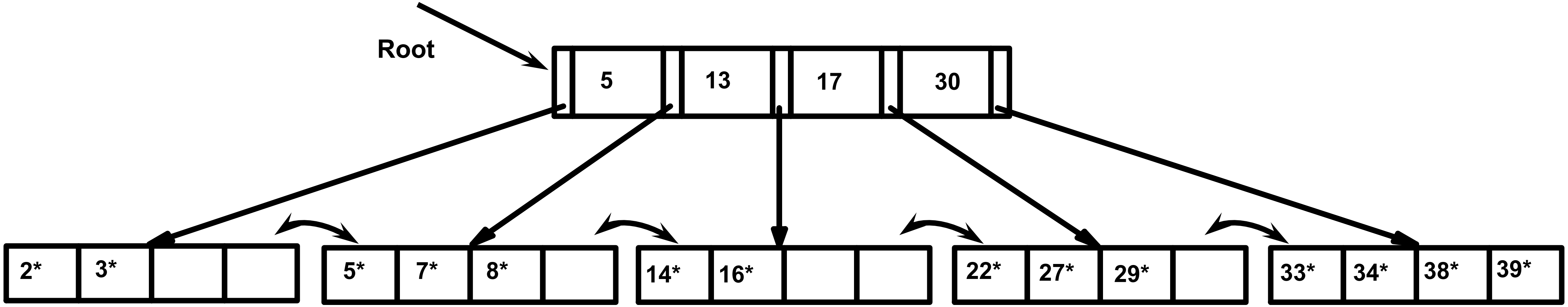


- Deleting 19* is easy.
- Deleting 20* is done with re-distribution. Notice how middle key is *copied up*.

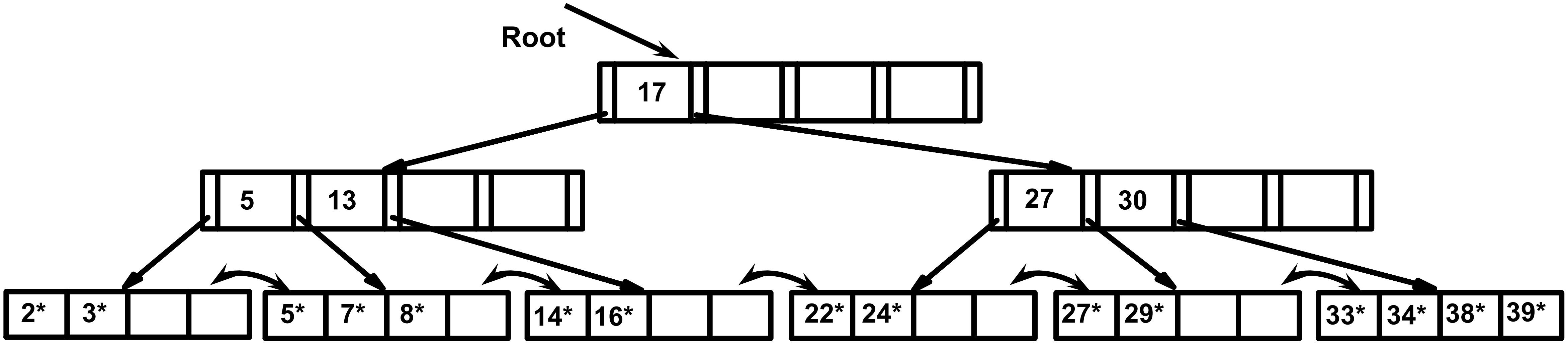
... And Then Deleting 24*

- Must merge.
- Observe *`toss'* of index entry (on right), and *`pull down'* of index entry (below).



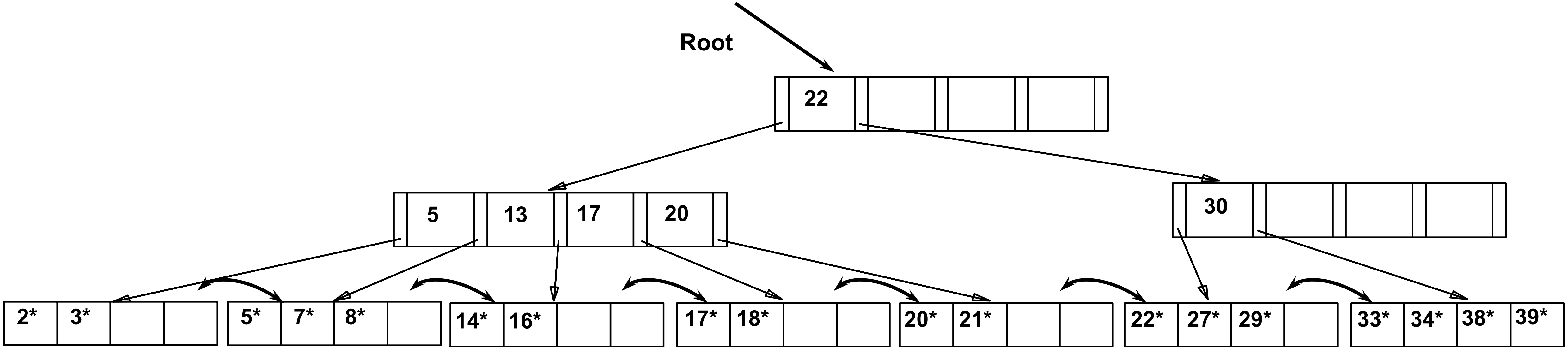


Example Tree Delete 24* ...



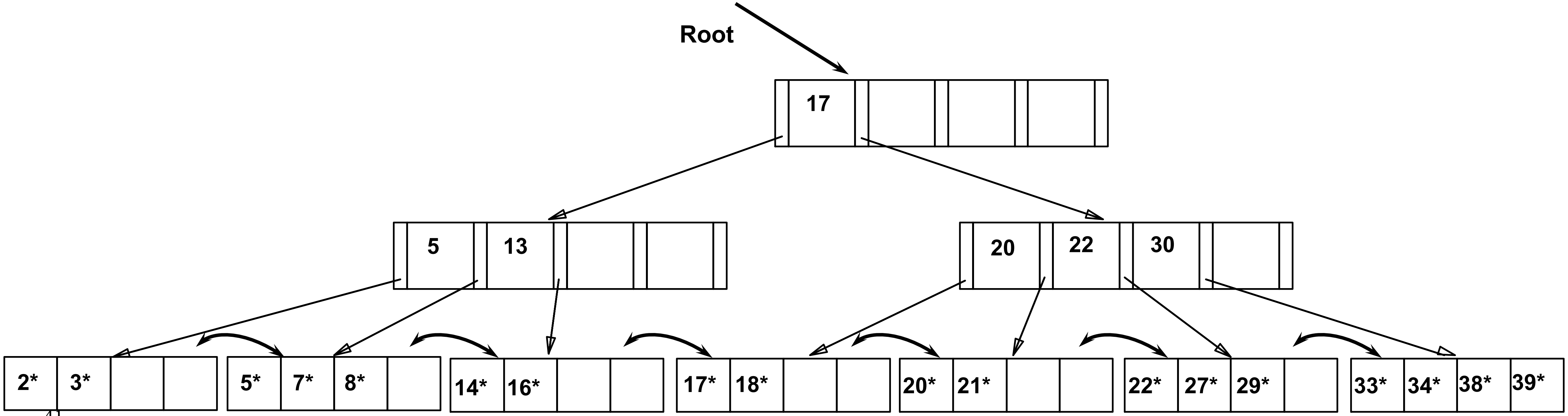
Example of Non-leaf Re-distribution

- Tree is shown below *during deletion* of 24*. (What could be a possible initial tree?)
- In contrast to previous example, can re-distribute entry from left child of root to right child.



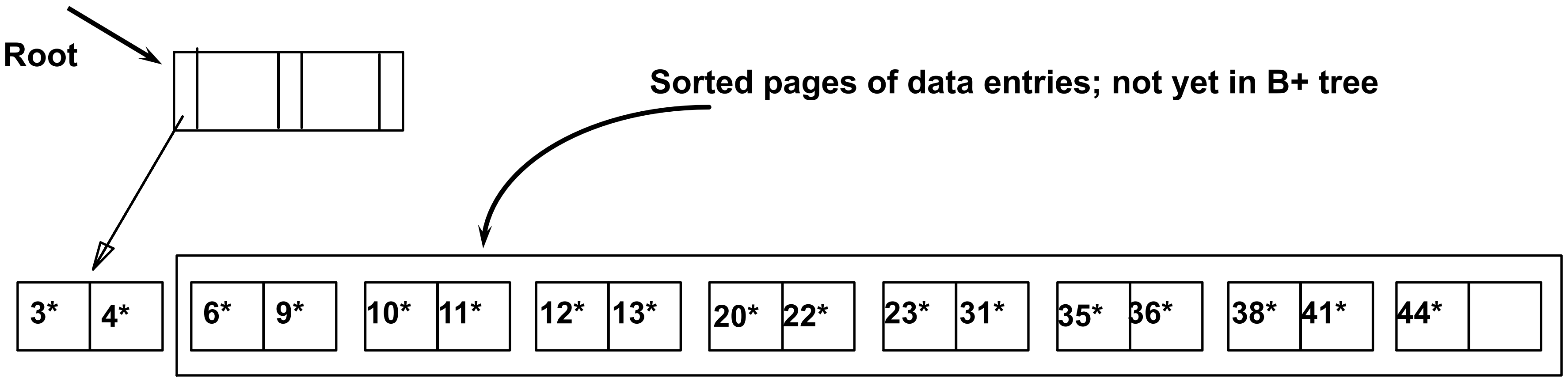
After Re-distribution

- Intuitively, entries are re-distributed by `*pushing through*' the splitting entry in the parent node.
- It suffices to re-distribute index entry with key 20; we've re-distributed 17 as well for illustration.

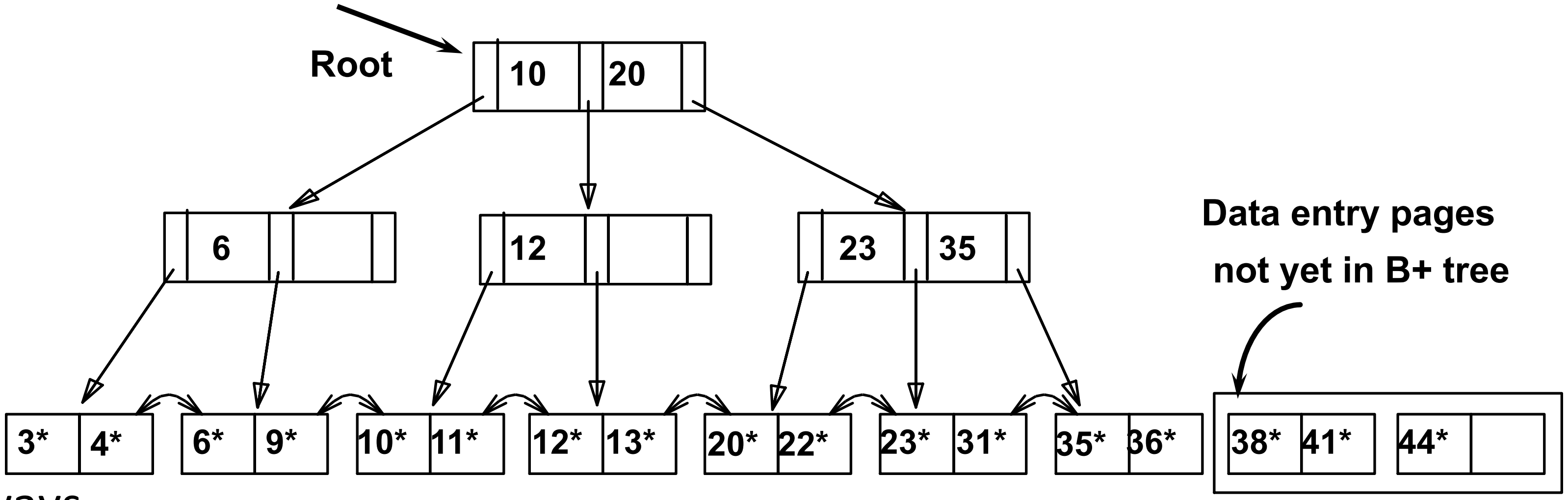


Bulk Loading of a B+ Tree

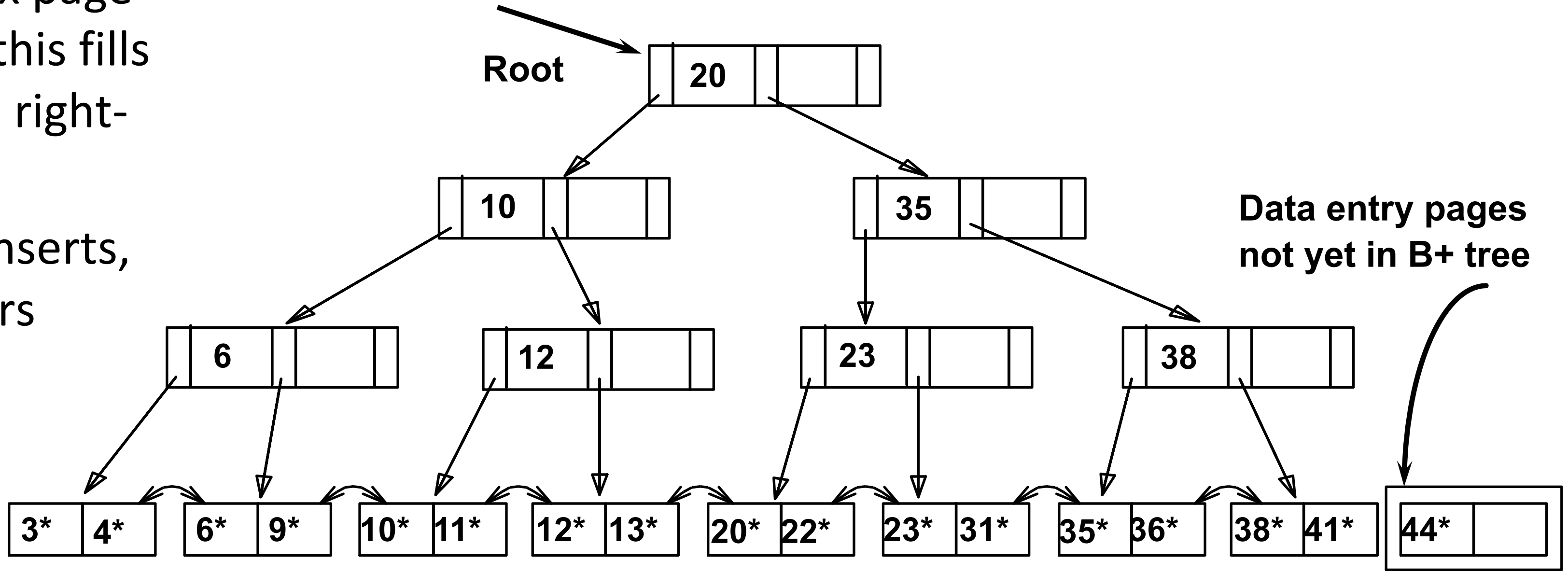
- If we have a large collection of records, and we want to create a B+ tree on some field, doing so by repeatedly inserting records is very slow.
  - Also leads to minimal leaf utilization --- why?
- *Bulk Loading* can be done much more efficiently.
- *Initialization*: Sort all data entries, insert pointer to first (leaf) page in a new (root) page.



Bulk Loading (Contd.)



- Index entries for leaf pages always entered into right-most index page just above leaf level. When this fills up, it splits. (Split may go up right-most path to the root.)
- Much faster than repeated inserts, especially when one considers locking!



Summary of Bulk Loading

- Option 1: multiple inserts.
  - Slow.
  - Does not give sequential storage of leaves.
- Option 2: *<u>Bulk Loading</u>*
  - Has advantages for concurrency control.
  - Fewer I/Os during build.
  - Leaves will be stored sequentially (and linked, of course).
  - Can control “fill factor” on pages.

A Note on `Order’

- *Order* (d) concept replaced by physical space criterion in practice (`*at least half-full*’).
  - Variable sized records and search keys mean different nodes will contain different numbers of entries.
  - Even with fixed length fields, multiple records with the same search key value (*duplicates*) can lead to variable-sized data entries (if we use Alternative (3)).
- Many real systems are even sloppier than this --- only reclaim space when a page is *completely* empty.

Summary

- Tree-structured indexes are ideal for range-searches, also good for equality searches.
- ISAM is a static structure.
  - Only leaf pages modified; overflow pages needed.
  - Overflow chains can degrade performance unless size of data set and data distribution stay constant.
- B+ tree is a dynamic structure.
  - Inserts/deletes leave tree height-balanced; $\log_F N$ cost.
  - High fanout (**F**) means depth rarely more than 3 or 4.
  - Almost always better than maintaining a sorted file.

Summary (Contd.)

- Typically, 67% occupancy on average.
- Usually preferable to ISAM, adjusts to growth gracefully.
- If data entries are data records, splits can change rids!

- Bulk loading can be much faster than repeated inserts for creating a B+ tree on a large data set.
- Most widely used index in database management systems because of its versatility. One of the most optimized components of a DBMS.